PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores

J. Eliot B. Moss[1], David S. Munro[2], and Richard L. Hudson[1]

Department of Computer Science[1]
University of Massachusetts
Amherst, MA 01003-4610, USA
{moss,hudson}@cs.umass.edu

Department of Computer Science[2]
University of St. Andrews
St. Andrews, Fife, KY16 9SS, UK
dave@dcs.st-and.ac.uk

Abstract. Traditional garbage collection techniques designed for language systems operating over transient data do not readily migrate to a persistent context. The size, complexity, and permanence characteristics of a persistent object store mean that an automatic storage reclamation system, in addition to ensuring that all unreachable and only unreachable data is reclaimed, must also maintain store consistency while limiting I/O overhead when collecting secondary-memory data.

Research has shown that careful selection of which area of a store to collect can significantly increase the amount of reclaimed storage while reducing the I/O costs. Many garbage collectors for existing stores, however, either are off-line or rely on reclaiming space in a predefined order. This paper presents a new incremental garbage collection algorithm specifically designed for reclaiming persistent object storage. The collector extends the Mature Object Space algorithm to ensure incrementality in a persistent context, to achieve recoverability, and to impose minimum constraints on the order of collection of areas of the persistent address space.

Keywords: persistence, garbage collection, memory management, mature object space, train algorithm

Introduction

The principal intention of automatic storage reclamation is to relieve the programmer of the burdensome and often error-prone task of indicating which memory can be reused. It can also help reorganize data in an effort to improve performance. To achieve this a collector must distinguish reachable from unreachable objects and reclaim the storage occupied by garbage. An important aspect of any collector is that it is *complete:* it will, after a finite number of invocations, reclaim all unreachable storage.

In a persistent store, automatic reclamation still needs to meet the criteria above. Indeed, lack of completeness will be a more acute problem than in transient systems since failure to reclaim all garbage leads to permanent space loss. Although many garbage collectors for main-memory programming languages and systems have been designed, built, and measured (see Wilson's survey [Wilson, 1992] for a good introduction), the collection of garbage for a persistent store raises additional concerns:

- The sheer size of many persistent stores suggests that semi-space techniques will be unworkable because they approximately double space requirements. Likewise, "stop-the-world" style collection would result in prohibitively long pauses.
- Because pointers in these stores refer to objects in secondary memory, updates resulting from object movement in copying and compacting collectors can incur high I/O costs.

- Persistent stores exhibit some notion of stability whereby a consistent state can always be reconstructed after a crash. Most existing collector algorithms are not inherently atomic and are thus unsuitable in this context.

We present here a new garbage collection algorithm for persistent object stores, called PMOS, tailored to address the issues above. The PMOS collector is essentially an extension of the Mature Object Space (MOS) algorithm [Hudson & Moss, 1992] (colloquially known as "the train algorithm") which is an incremental main-memory copying collector specifically designed to collect large, older generations of a generational scheme in a non-disruptive manner. There are a number of essential features of MOS that make it an attractive starting point:

- The collector limits the amount of data moved during each incremental invocation.
- It naturally supports compaction and clustering.
- It can be implemented on stock hardware and does not require special operating systems support such as pinning or external pager control.
- It has been implemented, proved to be complete, and to achieve the stated objective of bounded time for any single collection [Seligmann & Grarup, 1995].

The PMOS collector partitions the persistent address space into distinct areas and retains many of the same features and mechanisms of MOS, but with two important extensions:

- Unlike MOS, the algorithm does not impose any constraints on the order of collection of the areas. Work by Cook, Wolf, and Zorn [Cook *et al.*, 1994a, Cook *et al.*, 1994b] suggests that a flexible selection policy allowing a collector to choose which partition to collect can significantly reduce I/O and increase the amount of space reclaimed. One of the design goals for PMOS was to free the algorithm from imposing any collection order thus allowing the implementor to provide a policy appropriate to the application.
- Results from the MaStA I/O cost model work [Scheuerl *et al.*, 1995, Munro *et al.*, 1995] suggest that no single recovery mechanism gives the best performance under different workloads and configurations. Hence one of the aims of the PMOS design was to provide atomicity to the MOS collector without binding it to a particular recovery mechanism.

Related Work

There are two bodies of prior related work in garbage collecting databases and object stores. The first, older, body is concerned primarily with designing garbage collection schemes that will work in the concurrent and atomic world of databases (i.e., in the presence of concurrency, concurrency control, and crash recovery). See, for example, [Detlefs, 1990, Kolodner, 1987, Kolodner *et al.*, 1989, Kolodner, 1990]. The second, more recent, body is more concerned with policies and performance [Cook *et al.*, 1994a, Cook *et al.*, 1994b]. Our scheme differs from the first group in that it uses *partitions* (we call them blocks or cars) and uses them to collect in a coarse-grained incremental way, but at least somewhat obeying the existing object clustering pattern. Granted, the earlier schemes outlined above work on a page by page basis, but our partitions may be larger than a page and might vary in size, as convenient. Also, these earlier schemes tend to impose a specific order of collecting pages (breadth

first copying). The essential difference is that this earlier body of work was concerned with devising *correct* algorithms, in the face of concurrency and/or failures. We consider the correctness issue to be solved and are more concerned with issues of *performance* (but also portability, clean interfaces, etc.). Further, the correctness of recovery and concurrency control with our algorithm is simple to argue, whereas some previous schemes had more subtle, integrated algorithms.

The more recent body of work uses partitions, and is concerned with performance, but uses algorithms that do not guarantee completeness. Our scheme offers completeness, and allows (even requires in some sense) objects to be reclustered as they are collected. The work of Cook, Wolf, and Zorn has done a useful job in starting the investigation of suitable *policies* for selecting an order in which to collect partitions. To our knowledge, Bishop introduced the notion of partitioned gc [Bishop, 1977].

There are numerous papers on concurrent and/or distributed garbage collection, and undoubtedly more on persistent store or object base collection as well, but the references discussed above are representative of the prior and current art in object base garbage collection algorithms and performance.

Review of the Mature Object Space Algorithm

While generational schemes help reduce the length of time the average collection takes, the oldest generations tend to be large, and in any case, collecting them involves collecting all generations at once. The result is that while such collections are infrequent, they are unpleasantly slow when they do happen. Mature Object Space was designed to overcome this problem.

The basic idea of MOS is to divide the oldest generation into a number of f i xed size blocks, and to collect just one block at a time.[1] The hard part is to guarantee that all garbage is collected eventually. Completeness is achieved by organizing the collections in a way explained by using a metaphor: *trains* made up of *cars*.

Each block of MOS is a car, and each car belongs to a train. The cars of a train are ordered by age, with the oldest car at the "front" of the train. New cars are added to the "rear" or end of the train. It is assumed there are at least two trains (it is a policy issue as to the number of trains), and that the trains are also ordered from oldest to newest (in terms of the time the trains were created).

The goal of the MOS algorithm is to copy reachable data out of the oldest train into other trains, and then to discard the oldest train when it contains no reachable data. In this way, cycles of garbage larger than a single car can be reclaimed, *if* we can get them into a single train. At each collection, we copy all reachable data out of the oldest car of the oldest train. However, we are careful as to where we copy the data:

1. Data locally reachable [2] *from global variables, the program stack and registers, or from younger generations*, is copied to any other train, adding a car to that train if needed. The youngest train might be a good destination for the objects moved.

2. Data locally reachable *from other trains* is copied to those trains, adding a car if needed. If an object is reachable from more than one other train, it may be copied to any train from which it is reachable (one might pick the youngest train to put off copying the data again soon).

[1] The technique can handle objects larger than a block as well; see [Hudson & Moss, 1992] for details.

[2] To be precise, we say an object Y in the oldest car is locally reachable from a source object X if there is a direct pointer from X to Y, or there is a chain of pointers X, Y1, Y2, ..., Y where X has a pointer to Y1, Y1 has a pointer to Y2, etc., and all the Yi objects are in the oldest car.

3. Data locally reachable *from other cars of this train* is copied to the youngest car of this train, adding a new car as needed.

4. Remaining data is *unreachable* and is reclaimed immediately.

Note that the above steps are performed *in order*. Steps 1, 2, and 3 can be performed efficiently if we keep per-car remembered sets. One more rule we need is the following:

0. If no object in the oldest train is reachable from outside the train, reclaim the *entire train*. If necessary, create another train (to insure that there are always at least two trains).

The precondition of this rule can be determined by keeping a count of the number of references from outside the oldest train to objects inside it. This count is the sum of such counts for each car, and can be maintained fairly simply by keeping the count for each car and adjusting the total count after each collection.

Changes Needed to Support Object Store Collection

The MOS algorithm seems appropriate for garbage collecting object bases: it works a block at a time (what we called a *coarse-grained incremental* approach based on *partitions*), and it guarantees all garbage will be collected eventually (it is complete). However, as pointed out above, MOS is not suitable for object store garbage collection as it stands, principally because of the forced collection order and the I/O costs induced by pointer updates.

The MOS algorithm records only pointers from newer cars (and references from outside MOS) to older cars. Since only the oldest car is ever collected, this works out nicely in two ways. First, (only) the oldest car's remembered set information includes all references from outside that car. Thus, the knowledge required by the algorithm is available when needed. Second, pointers *from* the oldest car appear in *no* remembered set, and thus other cars' remembered sets never need to have items *removed*, only added.

In order to collect cars in any order (which is not particular to object base garbage collection but can also be used for main memory garbage collection), complete remset information is needed for *every* car, and some way of updating that information is required as cars are collected. Maintaining complete remset information for cars does not pose conceptual problems, but it does raise performance issues. For simplicity and performance when collecting, it would be preferable if a car's remset be stored with the car. However, keeping that remset accurate means fetching, updating, and writing back that remset any time a pointer to any object in the car is created or destroyed.

We propose to solve the problem in the following way. When a car is read in, its *outgoing* references (references to objects in other cars) are summarized. When a modified car is about to written back, its outgoing references are summarized again, and the differences recorded, namely references destroyed and references created. Note that collecting a car causes all of its outgoing references to be destroyed, and new references to be created (from other cars) from copied surviving objects. The remset changes are noted along with the car changes, and a table of changes not yet applied is maintained, called a Δ*ref set*. When a car is read in, any changes that are pending for that car's remset can then be applied. This process is illustrated in Figure 1.

Initially, an object in Car A refers to object o in Car B. The pointer is modified to refer to object o' in Car C, which results in two entries in the Δref set, one indicating that a reference from Car A to object o has been dropped, and one indicating that a reference from Car A to object o' has been added.

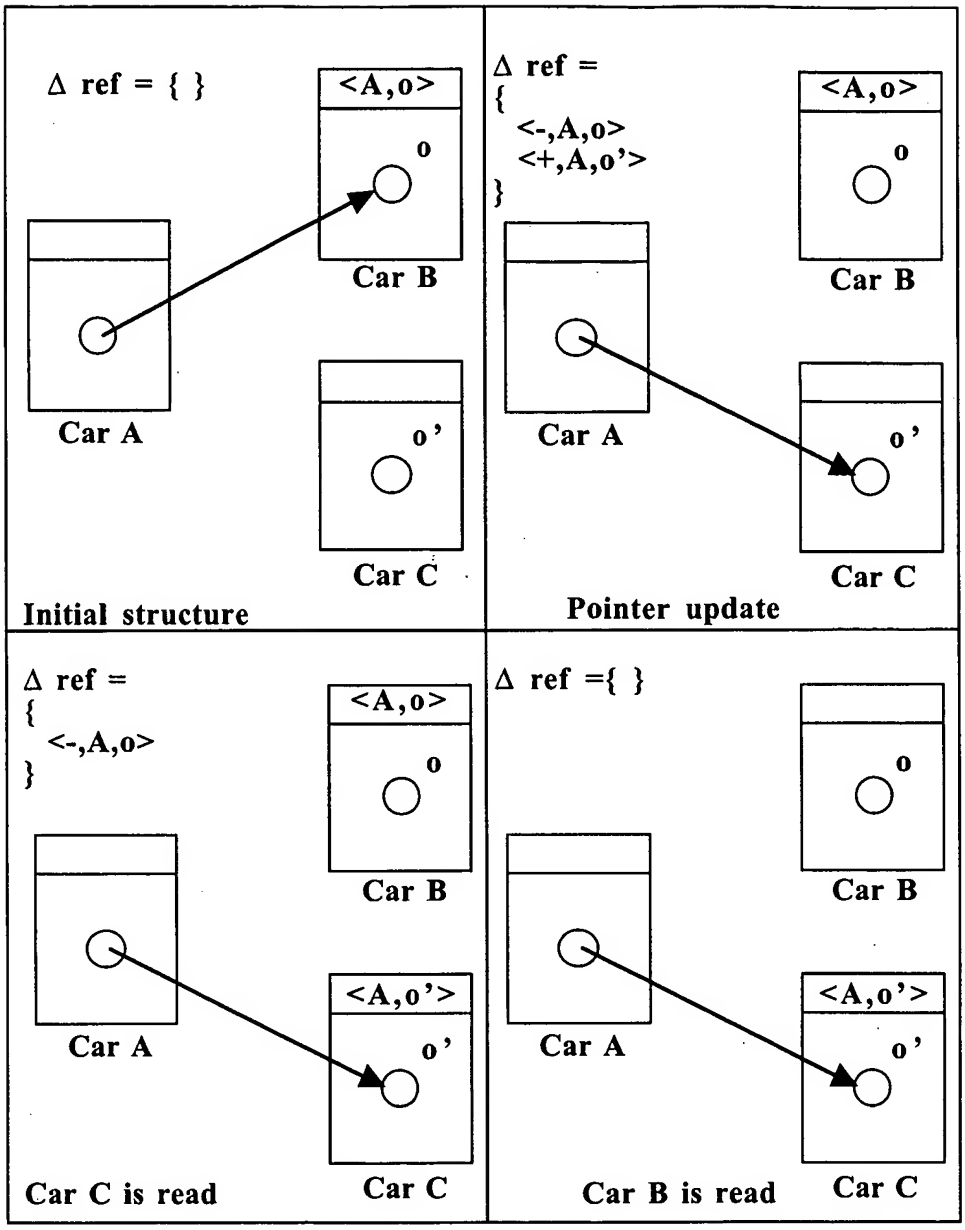

Figure 1: Illustration of algorithm handling creation/deletion of references

(Object "names" such o and o' are considered to include the identity of the car currently containing the object.) Note that the remsets for Car B and Car C are *not* updated yet. When Car C is read in, we update its remset and remove the + entry from the Δref set. Likewise, when Car B is read, we delete the remset item and the − entry in the Δref set. The point is that pointer changes require immediate access only to the car being modified and the Δref set, and actual remset additions and deletions can be deferred to a later time. The Δref set would probably be maintained in primary memory, and only moved to disk as a last resort. Presumably there would be background actions scheduled to read cars so as to purge the Δref set. It is now well known that such scheduled I/O is quite advantageous compared with synchronous I/O.

A similar process is followed for dealing with movement of objects from one place to another,

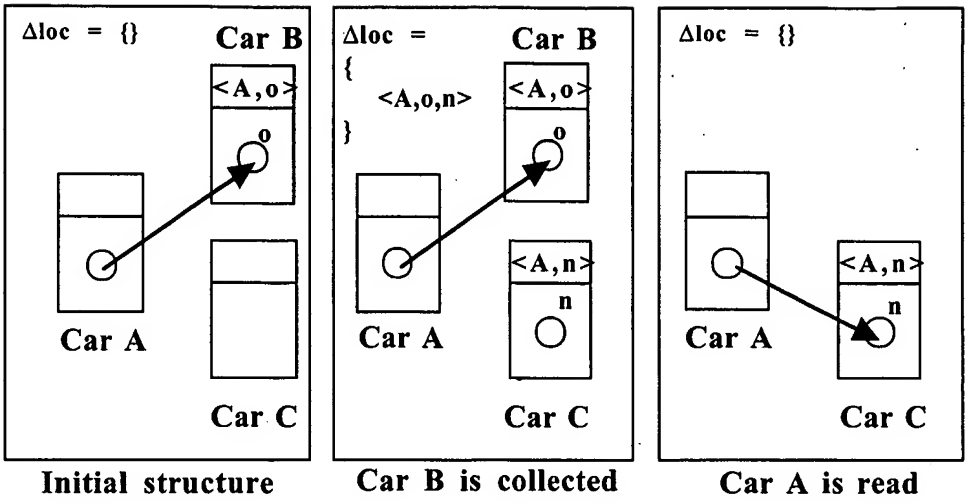

Figure 2: Illustration of algorithm handling movement of objects

using a Δ*loc set*, as shown in Figure 2. Here object o in Car B is moved to Car C when Car B is collected. The new object address is n, and we create an entry in the Δloc set indicating that Car A has a reference to object o which has now moved to location n. When Car A is next read in, the Δloc entry is applied, updating the pointer and " consuming" theΔloc entry. The principle behind the Δloc set is the same as for the Δref set: to allow updates to modify only locally available information, and to defer the remaining work to a later time.

The Δref and Δloc sets can be thought of as a special kind of log, and indeed, one could implement recovery of the Δref and Δloc information that way. However, it may be more convenient to use an underlying logging mechanism and to view the Δref and Δloc sets as just additional database data. In this way we achieve independence of any specific recovery mechanism. Likewise, when updating a car, we need only lock (or apply other appropriate concurrency control) that car, even even just the accessed subparts of the car. We do need to apply concurrency control to the data structures used for the Δref and Δloc sets as well, but their set semantics will allow high-concurrency semantics-based techniques to be applied, if necessary, to achieve the best performance.

The completeness of the original MOS algorithm substantially depends on the order in which the cars are collected. This comes about through the rule that places objects reachable from other trains into (one of) those trains. If cars are allowed to be collected in any order, completeness may be lost. For example, objects might just shuffle back and forth between two trains if the objects refer back and forth, and a cycle of such garbage objects may never be collected. The proposed solution is to maintain a notion of order (or age) of the trains, and allow objects to be copied from one train to another only in one direction. This is adequate for insuring completeness, if combined with a rule that every train has every car collected eventually. Note that this does not affect the choice of which car to collect, but may affect the degree of progress made in collecting any particular car.

The Persistent MOS Algorithm

We now present the PMOS algorithm itself. We will offer detailed pseudo-code in an online technical report, since space precludes presenting it here. For present purposes, we offer a description in English, at a somewhat greater level of detail than the overview given above.

Data Structures

The PMOS algorithms' data structures include a *store*, *trains*, *cars*, *objects*, *remsets*, a Δloc set, a Δref set, and *buffers*.

Stores: A store *s* consists primarily of a set of *trains*, used to group cars as in the original MOS algorithm, a set of *cars*, which contain the objects of the store and associated information for the algorithm (and each of which is associated with a train), and a set of *root objects*, designated by their *locations* in the store. An object location is a unique name for an object within the store. We assume that we can determine an object's car from its location, and thus access the object, but locations need not have any particular form (i.e., they may be physical addresses, object identifiers, etc.). A store also has an associated Δloc set and Δref set, and a set of buffers.

Trains: Trains are numbered 1, 2, 3, ..., with new trains assigned numbers higher than any existing train. We assume that we can readily determine the train of any car, and of any object; in practice this probably requires maintaining various tables and keeping them resident in main memory insofar as possible. Each train also has associated with it an *external reference count*, summarizing the number of references from outside the train to objects stored in cars of the train; this includes root references (members of the containing store's root object set). Each train also has a set of zero or one *old root references*, and zero or one *old cross-train references*, whose use is explained later.

Cars: Every car has an associated identifier, unique among the cars of a store. We assume we can find and retrieve a car and its associated information given the car's identifier. A simple way to assign car identifiers is from a persistent global counter, but any scheme insuring uniqueness will work. We assume that it is possible to examine the objects in a car and determine the references they contain to objects in other cars. These are termed *outgoing references*. While a car may contain (objects having) multiple references to the same object, each reference occurs only once in the outgoing references set. We note that any given object is contained in only one car at a time.

Objects: An object is identified by its unique location (or "name"). We are not much concerned with the internal structure of objects, except that we assume we can find and update outgoing pointers from any given object, and so we ignore details of object format, copying their contents, etc.

Remsets: Every car has an associated remembered set, containing information about *incoming references*. Each entry in a remset identifies the object that is the target of the reference and the car containing the source of the reference. If a car has multiple references to the same object, they are summarized by a single remset entry.

Δloc sets: A Δloc set records information about objects that have moved, so that references to those objects can be updated at some later time. The algorithms maintain a single Δloc set for the entire store, summarizing all object references that need updating. We use separate entries for each car needing updating to avoid the problem of knowing when to discard object relocation information if all we recorded was the old and new address of each object.

Δref sets: A Δref set records information about cross-car references that have been created or deleted. (A pointer change is treated as a deletion plus a creation.) This information is used to update the remset of a target car at some later time. The algorithms maintain a single Δref set summarizing all remset updates that are pending.

Buffers: A buffer holds a main memory version of a car, so it has all the data structures that the car has. Additionally, each buffer has an *old outgoing reference set*, which summarizes the car's outgoing pointers as they were when the car was read in (an empty set for a newly created car). Note that buffers are lost in case of a system failure.

For the moment we will assume that each car's remset can be stored with the car, and that the Δloc

set and Δref set fit in main memory. We will discuss later how we can relax these assumptions.

Algorithms

We describe how PMOS performs a number of operations on the store, each in turn. For brevity we omit a number of simple and obvious operations, such as initializing new stores, trains, and cars.

Adding a root: This must add the root to the store's root set and increment the external reference count of the train containing the new root object.

Removing a root: We remove the root from the store's root set and decrement the external reference count of the train containing the deleted root object.

Reading in a car: We create a buffer, read in the car, summarize and save the car's outgoing reference set, and then update in the buffer the car's references and remset (separate algorithms below).

Updating a car's references: This consists of locating any Δloc entries for this car and applying them to each object reference in the buffer, and in the saved outgoing reference set. The applied Δloc entries are deleted from the Δloc set.

Updating a car's remset entries: For each Δref entry indicating that a new remset entry needs to be added, add the entry and delete the Δref item. Likewise, for each Δref entry indicating that an existing remset entry should be deleted, delete the entry from the remset and delete the Δref item from the Δrefset.

Writing a buffer back to the store (updating a car): Scan the buffer and build a new outgoing reference set. Compare it with the saved outgoing reference set. For old items that no longer appear, add a Δref item indicating that this car no longer refers to the target object. Similarly, for new items that did not previously appear, add a Δref item indicating that this car now refers to the target object. Then we write the car to persistent storage.

Garbage collecting a buffer: For objects locally reachable from roots, move them to any car in a higher numbered (younger/newer) train. For objects locally reachable *from higher numbered trains*, move them to any car of a referring train. For objects locally reachable from other cars of the current train, or only from lower numbered (older) trains, move the objects to another car of the current train. The preceding steps must be applied in order; see below for details of adjustments to data structures when objects move. All other objects are unreachable and can be discarded. We use the saved outgoing reference set to generate new Δref entries to indicate that the references from this car are now gone. The car and buffer can now be deleted. Note that garbage collecting does require having present all the cars to which we are moving objects. With additional data structures, we could define a holding set for objects " in transit" and avoid this requirement, but we are not certain is is a good idea, because it may place too high a demand on primary memory; yet it may be worthy of further investigation.

Moving an object from one car to another: Let us call the original location of the object its *source car* and its new location the *destination car*. We allocate space in the destination car and copy the object's contents. For each remset entry of the source car pertaining to the moved object, we delete the remset entry from the source car and add a corresponding entry to the destination car. If the cars are in different trains, we need to decrement the source train's external reference count appropriately, and increment the destination train's external reference count. Whether a particular remset entry causes a decrement (increment) depends on whether the source (destination) train also contains the car referring to the object being moved. For each remset entry we must also add an item to the Δloc set.

Additionally, we must also insure that we update any references from resident buffers to the moved object, that we update the root set if the object being moved is a root, and that we update the Δloc set if there remain pending entries from a previous move of the object.

Collecting a whole train: If a train's external reference count goes to zero, we can collect the entire train. However, we must make sure that any resident cars of the train are entirely up to date, since in-memory modifications may have created pointers, etc.

Insuring progress: The algorithms described above do not absolutely insure progress. As with the original MOS algorithm, a running program can change pointers in between collections and prevent the algorithm from removing objects from a given train. We give a series of necessary changes here, to avoid cluttering the descriptions of the algorithms offered above.

What is needed is to record for each train, one external reference. It may be an *old root reference* or an *old cross-train reference* (from a higher numbered train only). Whenever we create a new root (cross-train) reference to a given train, we save it as the "old" reference if we do not already have an "old" reference saved. When we collect a car in the train, if there are no current root or cross-train references that allow us to move objects from the car, then we check the train's "old" root and cross-train references. If either mentions an object in the current car, we move that object (as for a regular root or cross-train reference). If there *are* references that allow us to move at least one object out of the current train, we clear out the train's old root and cross-train references, since their only purpose is to guarantee progress and we made progress.

Correctness Argument

Now that we have described the algorithm, we sketch a correctness argument, in several stages:

1. **Remset Invariant:** That remsets, once updated from Δref, reflect exactly all cross-car references.
2. **Location Invariant:** Object references, once updated from Δloc, correctly reflect the object graph.
3. **Train Count Invariant:** Train counts accurately reflect the number of cross-train plus root references. In particular, if a train's count is zero, then no objects in it are reachable.
4. **Safety:** That no object reachable from a root is ever collected.
5. **Completeness:** That if every car is collected eventually, then all garbage is collected eventually.

Remset Accuracy

The invariant is easy to establish initially. There are four changes to the world that we must consider:

1. Adding a cross car reference: This is handled by the algorithm for writing a modified car back to the store. It detects a new reference and adds it to the Δref set. If the target car is resident, we will update the remset before writing the car back; if the car is not resident, it will be read in eventually and the remset update applied by the car reading algorithm at that time.
2. Deleting a cross car reference: This is handled analogously to adding a reference, using a $-$ entry rather than a $+$ entry in the Δref set.
3. Moving an object that is a target of cross-car references: In this case the objet moving algorithm populates the new object's car's remset from the remset of the old car, and the old remset entries are discarded, which correctly accounts for the change.

4. Moving an object that contains cross-car references: When we write back the car containing the new object, we will detect the new references and add any necessary new remset entries. The car collecting algorithm takes care of noting that the old remset entries need to be deleted.

Reference Accuracy

The invariant is trivially established initially. The only thing that can affect it is movement of objects, which inserts entries into Δloc based on the old object's car's remset entries (which we just argued are correct). For a non-resident referring car these entries are applied when the car is next read in. We have several options as to how to update any references that are currently resident (immediate, deferred, etc.), but we assume they are taken care of some time before the resident cars are written back. Things still work if an object should move multiple times before a referring car is read in, because the object moving routine also takes care of updating pending Δloc entries.

Train Count Accuracy

Again, the invariant is easy to establish initially. The root manipulating algorithms adjust the count in the obvious way, and the car writing algorithm adjusts counts in exactly those places that references are seen to be created or destroyed (and those references are from one train to another).

Safety

There are two ways in which objects are discarded: from single car collections and from the dropping of entire trains. Given that train counts are correct as just argued, dropping a train whose count is 0 is correct, since a 0 count means the train has no root objects and there are no references from other trains to objects in it. Hence there is no path from any root object to any object in the train. (Note that we must take care that any resident cars do not have unrecorded references to objects in the train; hence, we may need to "synchronize" our knowledge of the contents of resident cars.)

Correctness of car collection is also fairly straightforward. We move, and thus preserve, all objects in that car reachable from roots (and "relocate" those roots). We also move all objects reachable from outside the car. Hence, anything left is not reachable from outside the car and is garbage.

Completeness I: Progress Evacuating the Lowest Train

One way in which the algorithms could fail to collect all garbage eventually is if they do not make progress. While we require that each car be collected eventually, and (as we shall see later) the algorithm is complete as described so far, this is true only provided that the mutator (running program) does not move pointers and roots around in between car collections. The MOS algorithm has the same problem, noted and fixed by Grarup [Seligmann & Grarup, 1995]. Here is a description of the problem via an example. Suppose we have a cycle of objects lying in two cars, and one of the objects is a root. If we collect both cars, we would expect to remove all the objects to another train, but consider the following sequence of actions. We collect the car not containing the root (the objects stay in the current train since they are reachable from the car with the root). The mutator moves the root to the car we just collected. We now have a picture like before, which can perpetuate indefinitely.

The solution is to remember an "old root" and treat it as indeed a root. This old root reference it guarantees that if the mutator creates a root, and hence some object was reachable by the mutator at that time, we will move the object out of the train later. If the object becomes garbage, then it is not

a root when it is moved, and in any case, it is not added to the new train's old root set (not the least because, since the object is garbage, the mutator cannot access it to designate it a root). Thus the old root technique will not cause garbage to be retained forever.

Similar to moving roots around, the mutator could move around pointers from some other train. The old cross-train reference solves this problem in the same way that the old root reference does for roots. There is one difference, though: we need worry only about references from higher numbered trains, since we cannot move an object to a lower numbered train.

We arrive at the following progress argument. Consider the set of cars of the lowest numbered train at some time *t*. Eventually, we will collect each of those cars. Either the train's count has gone to 0, and the entire train has been collected (progress) or the count is not 0. If the count is not 0, and the train had an "old" reference (root or cross-train) at *t*, then we will have move at least the "old" object to another train, reducing the number of objects in the lowest numbered train. If the train had no "old" object at *t* and did not gain any, then we will have encountered each of the root and cross-train references in the original count, and thus will have made progress. If there was no "old" reference, we gained one, and now have none, then we made progress, since the "old" references are cleared out only when we make progress. Finally, if we have an "old" reference, when we process the car referred to, we will make progress. Thus, from any point in time, there will always be a future time when the lowest numbered train will have its count go to 0 or we will remove an object from it. The one additional thing we need is for the mutator not to allocate new objects in the lowest numbered train - then we can guarantee that the lowest numbered train will always shrink.

Completeness II: Progress Reclaiming Garbage

Suppose a particular object *o* becomes unreachable from any root. We argue that it will eventually be reclaimed. First, if *o* was mentioned as an "old" object from a time when it was still reachable, we will move *o* once; thereafter, it can never be an "old" object again (because the mutator cannot refer to it, since it is unreachable). Now consider the case in which no object refers to *o*. If *o* is "old", it can be moved once, but after that it will be collected in a local collection. Thus, trees and DAGs of garbage will disappear from their roots down, since each object will eventually not have any references to it.

That leaves only cycles of garbage for consideration. First, we know that eventually each car containing the cycle will be collected, and thus any "old" references from the time before the cycle was garbage will be processed. Once that has happened, consider an object of the cycle that is in the highest numbered train containing any object of the cycle. None of the cycle objects can be moved to any higher numbered train than that, and, by the progress argument, lower numbered trains are eventually evacuated, so in the worst case the cycle will eventually lie entirely in the lowest numbered train. There will be no "old" references to any of the cycle's objects, and so once all reachable objects are removed from this train, the train's count will be 0 and the cycle will be reclaimed.

Recovery Issues

We assume that there is some underlying serializable transaction system. in this view, a transaction updates cars, and as a "side-effect" creates Δref and Δloc entries. Note that collecting a car can be viewed as a transaction, too. The main point is that just as the changes to cars need to be recorded atomically, so also must we record changes to the Δloc and Δref sets. While we may wish to apply "special" (i.e., semantics based concurrency control) techniques to take advantage of the maximum possible concurrency, there is no need to develop specialized algorithms to permit garbage collection.

It *is* possible that garbage collecting will introduce additional concurrency control conflicts, because of its need to update cars that receive additional objects. However, with suitably sophisticated semantics based concurrency control, we can avoid most such conflicts while introducing only a moderate amount of additional implementation complexity.

Extensions

The PMOS algorithm as described assume that remsets never overflow cars and that the Δloc and Δref sets can be maintained in primary memory. How can we relax these restrictions?

Large Δref and Δloc sets

It is not too difficult to handle large Δref or Δloc sets. One technique is to store them in a suitably sorted secondary memory structure (e.g., a B-tree), with a caching mechanism to hold recently accessed chunks in primary memory. This induces extra overhead in some cases. Note that we can keep recent updates in a primary memory buffer before adding them to the secondary memory structure, and thus batch our updates to that structure.

An alternative strategy is to schedule affected cars to be read in, so that we can apply the pending updates and remove them from the Δloc and Δref sets. This approach seems promising in that the costs of scheduled I/O are likely to be substantially less than the synchronous reads introduced by the secondary memory data structure approach.

Large remsets

Large remsets may be more difficult to handle well. One needs to handle an overflow immediately. In such "emergency" situations a memory resident overflow table sounds like a good alternative. One could also add to the affected car a pointer to a remset overflow area stored elsewhere in secondary storage. In any case, the car will probably need to be marked as having overflowed. A combination of main memory and secondary memory overflow tables can probably be tuned to work well, given more knowledge than we currently have as to the distribution of the number of remset entries needed by each car. Note that it is not a good idea to attempt the " popular object" ideas of the MOS algorithm: the MOS algorithm's collection order can discard and rebuild remsets, whereas PMOS really needs them maintained at all times. Finally, it may be reasonable to consider *splitting* a car into two cars, each of which may then have a smaller remset. This will not work if the large number of references are all to one particular object. In that case we really must employ an overflow mechanism.

Conclusions

We have presented a new database / persistent store garbage collection algorithm, PMOS (Persistent Mature Object Space), and sketched arguments as to its correctness. PMOS is incremental (at the level of blocks of memory) and is guaranteed to collect all garbage eventually, provided only that each block is eventually collected individually. While PMOS is based on copying techniques, it does not require semi-spaces: it only needs enough space to copy the block being collected, and then that block will be freed. The design of PMOS is such that one can use any serializable concurrency control and recovery scheme to support it in a resilient multi-user environment. No prior schemes had these attributes of

incrementality, completeness, and independence of specific transaction mechanisms. We look forward to implementing PMOS and evaluating its performance in practice.

References

[Bekkers & Cohen, 1992] Yves Bekkers and Jacques Cohen, Eds. *International Workshop on Memory Management* (St. Malo, France, September 1992), no. 637 in Lecture Notes in Computer Science, Springer-Verlag.

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Cook *et al.*, 1994a] Jonathan E. Cook, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. Effectively controlling garbage collection rates in object databases. Tech. Rep. CU-CS-758-94, Department of Computer Science, University of Colorado, Boulder, CO, October 1994.

[Cook *et al.*, 1994b] Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)* (Minneapolis, MN, May 1994), pp. 371-382.

[Detlefs, 1990] David L. Detlefs. Concurrent, atomic garbage collection. Tech. Rep. CMU-CS-90-177, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1990.

[Hudson & Moss, 1992] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In [Bekkers & Cohen, 1992], pp. 388–403.

[Kolodner, 1990] Elliot Kolodner. Atomic incremental garbage collection and recovery for a large stable heap. In *Proceedings of the Fourth International Workshop on Persistent Object Systems* (Martha's Vineyard, Massachusetts, September 1990), Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, Eds., Published as *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990, pp. 185-198.

[Kolodner *et al.*, 1989] Elliot Kolodner, Barbara Liskov, and William Weihl. Atomic garbage collection: Managing a stable heap. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, June 1989), *ACM SIGMOD Record 18*, 2 (May 1989), pp. 15-25.

[Kolodner, 1987] Elliot K. Kolodner. Recovery using virtual memory. Tech. Rep. MIT/LCS/TR-404, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1987.

[Munro *et al.*, 1995] David S. Munro, Richard C. H. Connor, Ron Morrison, J. Eliot B. Moss, and Stephan J. G. Scheuerl. Validating the MaStA I/O cost model for DB crash recovery mechanisms. Position paper for workshop on database performance at OOPSLA '95, October 1995.

[Scheuerl *et al.*, 1995] Stephan J. G. Scheuerl, Richard C. H. Connor, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Validation experiments for the MaStA I/O cost model. In *Second International Workshop on Advances in Databases and Information Systems (ADBIS '95), Volume 1* (Moscow, Russia, June 1995), pp. 165-175.

[Seligmann & Grarup, 1995] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)* (Aarhus, Denmark, August 1995), no. 952 in LNCS, Spring-Verlag, pp. 235-252.

[Wilson, 1992] Paul R. Wilson. Uniprocessor garbage collection techniques. In [Bekkers & Cohen, 1992].